Hardware Automated Dataflow Deployment of CNNs

Technical Report Haddoc/2016-04TR01

K.Abdelouahab[1], M.Pelcat[1,2], J.Serot[1], F.Berry[1], C.Bourrasset[3], and J.C.Quinton[4]

[1]Institut Pascal,Clermont Ferrand, France
[2]IETR/INSA, Rennes, France
[3]CEPP Bull, Montpellier, France
[4]Laboratoire Jean Kuntzmann, Université Grenoble-Alpes,Grenoble, France

April 2017

Abstract

Deep Convolutional Neural Networks (CNNs) are the state of the art systems for image classification and scene understating. However, such techniques are computationally intensive and involve highly regular parallel computation. CNNs can thus benefit from a significant acceleration in execution time when running on fine grain programmable logic devices. As a consequence, several studies have proposed FPGA-based accelerators for CNNs. However, because of the huge amount of the required hardware resources, none of these studies directly was based on a *direct* mapping of the CNN computing elements onto the FPGA physical resources. In this work, we demonstrate the feasibility of this so-called *direct hardware mapping* approach and discuss several associated implementation issues. As a proof of concept, we introduce the HADDOC2 open source tool, that is able to automatically transform a CNN description into a platform independent hardware description for FPGA implementation.

1 Introduction

Convolutional Neural Networks (CNNs) [19] have become a *de-facto* standard that increased the robustness and accuracy of machine vision systems. It is possible nowadays to build high performance image classification systems by deploying large-scale, pre-trained CNNs models. However, this accuracy comes at the price of a high computational cost as state of the art CNNs may require up to 40 GOPs to classify a single frame [3]. As a result, implementing CNNs with real-time constraints is challenging task. A possible way to address this challenge is to take advantage of the massive fine grain parallelism offered by FPGA devices to embody the large amount of intrinsic parallelism exhibited by CNN-based algorithms. In this case, the problem boils down to find an adequate and efficient mapping between the computation model of the latter and the execution model supported by the former. Based on our previous experience in the implementation of real-time vision applications on FPGA-based platforms [23], we advocate the use of a *stream-based dataflow* model to solve this mapping problem. In this approach, a CNN-based algorithm is described as graph of dataflow actors exchanging data through unidirectional channels and this graph is statically and physically mapped onto the target FPGA using a library of pre-defined computing elements to implement actors.

In the sequel, we demonstrate the feasibility of this so-called *Direct Hardware Mapping (DHM)* approach for implementing realistic CNN-based applications onto Field-Programmable Gate Arrays (FPGAs). Moreover, we introduce HADDOC2, a software framework providing a fully automated implementation path for CNNs onto FPGAs using the DHM approach. The HADDOC2 tool is compatible with the widely used Caffe deep learning framework [16] and generates platform independent synthetizable VHDL code. In other words, we introduce in this work a tool that automatically maps a Caffe pre-trained model onto an FPGA device.

2 CNNs : Computations and parallelism sources

CNNs are a category of feed forward artificial neural networks that are bio-inspired by the visual cortex of the brain. The huge improvement of CNN-based algorithms was made possible by two factors: On one hand, the availability of massive-sized annotated image data-sets [8] allowed to train robust large scale feature extractors and accurate classifiers. On the other hand, the growth of high performance processors and, especially Graphics Processing Units (GPUs), provided the computational power required to train deeper and more complex neural networks [5]. A typical CNN structure, as shown in figure 1, will perform a succession of convolutions interspersed with sub-sampling layers. The last stages include typically two or three fully connected neural network for classification tasks. The depth (number of layers) of a CNN ensures better accuracy and less over-fitting and recent networks are usually very deep with 8 to 19 layers [25].

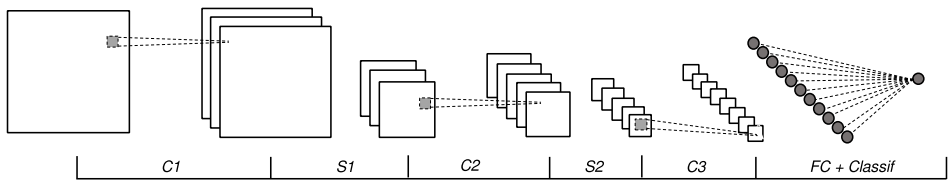

Figure 1: An example of a CNN topology with 3 convolutional layers (C1,C2,C3) two subsampling layers and one fully connected stage (FC).

2.1 Convolution layers

Convolutional layers are the most computationally intensive and are responsible – in a typical implementation – for more than 90% of the CNN execution time [7]. Each layer (l) extracts N feature maps from C input channels by performing N convolutions of size $K \times K$ on each input. This filtering is followed by the application of a non-linear activation function act and a bias term b_n to each set of features. As shown in equation 1, $N \times C$ convolutions are required to process a given layer.

$$
\begin{aligned}
&\forall l = 1 : L \textit{ (Number of conv layers)} \\
&\quad \forall n = 1 : N \textit{ (Number of output feature maps)} \\
&\quad\quad \forall i = 1 : Ix \textit{ (feature map rows)} \\
&\quad\quad\quad \forall j = 1 : Iy \textit{ (feature map columns)} \\
&\quad\quad\quad\quad f^{(l)}[n,i,j] = b^{(l)}[n] + \sum_{c=1}^{C} \sum_{p=1}^{K} \sum_{q=1}^{K} \Phi^{(l)}[c, i+p, j+q].w^{(l)}[n,c,p,q]
\end{aligned} \tag{1}
$$

where

- $\boldsymbol{f^{(l)}}$ is a tensor of output feature maps of layer (l)
- $\boldsymbol{b^{(l)}[n]}$ is the bias term applied to applied to feature n
- $\boldsymbol{\Phi^{(l)}}$ is a tensor of input feature maps of layer (l)
- $\boldsymbol{w^{(l)}}$ is tensor of pre-learned filters

As already pointed out in [20], the computations described in equations 1 exhibit a large amount of potential parallelism:

- **Inter Layer parallelism:** CNNs have a feed-forward hierarchical structure consisting of a succession of data-dependent layers. Layers can therefore that can be executed in a *pipelined* fashion where the execution of layer (l) can start before the execution of layer ($l-1$) ends.
- **Inter neuron parallelism:** Each neuron of a layer is independent when processing features. Thereby, a full data-parallelism can be exploited when computing concurrently each of the $N^{(l)}$ element of equation 1
- **Inter convolution parallelism:** All of the convolutions performed by a single neuron can also be evaluated simultaneously by computing concurrently the $C^{(l)}$ convolutions of equation 1.
- **Intra convolution parallelism:** 2D image convolution can be implemented in a pipelined fashion [24] allowing the $K \times K$ multiplications to be computed concurrently in equation 1

2.2 Subsampling layers

A common operation when conceiving CNNs is to periodically insert subsampling (or pooling) layers in-between successive convolutional layers. These downsample the inputs by selecting the *average*, or, more commonly, the *maximum* of a given neighborhood of each pixel as described in equation 2

$$
\begin{aligned}
&\forall l = 1 : L \text{ (Number of pool layers)} \\
&\quad \forall n = 1 : N \text{ (Number of output feature maps)} \\
&\quad\quad \forall i = 1 : Ix \text{ (feature map rows)} \\
&\quad\quad\quad \forall j = 1 : Iy \text{ (feature map columns)} \\
&\quad\quad\quad\quad f^{(l)}[n,i,j] = \max_{p,q \in [1:K]} \left(\Phi^{(l)}[n, i+p, j+q] \right)
\end{aligned} \tag{2}
$$

Pooling layers reduce the amount of parameters required to process the next stages of the network, which controls overfitting in one hand and decrease the computation load on the other.

2.3 Fully connected layers

A Fully Connected (FC) neural network –with usually 3 or 4 hidden layers– terminates CNNs and acts as a classifier. In this case, no parameters are shared across the feature-maps (feature maps and learned parameters have the same dimension). In this case, FC layer activations are computed with the inner product operation followed by a bias offset as detailed in equation 3

$$
\begin{aligned}
&\forall l = 1 : L \text{ (Number of FC layers)} \\
&\quad \forall n = 1 : N \text{ (Number of output feature maps)} \\
&\quad\quad \boldsymbol{f^{(l)}[n]} = \text{act} \left[b^{(l)}[n] + \sum_{c=1}^{C^{(l)}} < \boldsymbol{\phi^{(l)}[c]}, \boldsymbol{w^{(l)}[n,c]} > \right]
\end{aligned} \tag{3}
$$

where $<,>$ denotes the the inner product operator.

3 Direct Hardware Mapping of CNN entities

3.1 Dataflow processing of CNNs

The foundations of dataflow Models of Computation (MoC) were formalized by [9] in order to create an architecture where multiple fragments of instructions can process simultaneously a stream of data. Programs respecting dataflow semantics are described as a *network* (graph) of fundamental processing units commonly called *actors* and communicating abstract data messages called *tokens* on unidirectional First-In First-Out (FIFO) channels.

In terms of architecture-application matching, the CNN's layout fits naturally with a stream-based model of computation. In other words, the operations involved in feed forward propagation of a CNN –described in the latter section– can be executed following the stream-based dataflow MoC. In fact, CNN-based algorithms can be modeled as modeled as dataflow process networks (DPNs) where nodes correspond to processing actors and edges correspond to communication channels. Each actor follows a purely data-driven execution model where execution *(firing)* is triggered only by the availability of input operands.

The DHM approach consists of *physically* mapping entirely graph of actors onto the target device. Each actor becomes a computing unit with its specific instance on the FPGA and each edge is mapped to a signal.

3.2 DHM of Convolution layers

As stated in section 2.1, convolutional layers are the most computation intensive tasks in a given network. However, DHM approach fully exploits all the parallelism sources of theses layers. All neurons of a layer are mapped on the device to take advantage of intra-neuron parallelism (Fig 2-a). In neurons, each convolution is mapped separately (Fig 2-b) and finally, within a convolution engine, each multiplier is instantiated separately (Fig 2-c). As an example, figure 3 illustrates how a convolution layer C1 ($C = 3, N = 5, K = 3$) extracts 5 features from a 3-channel input pixel flow. In this example, 15 convolution and 5 activation blocks are mapped onto the FPGA as a result of the layer graph transformation, which corresponds to 135 multiplications, 20 summations and 5 activations.

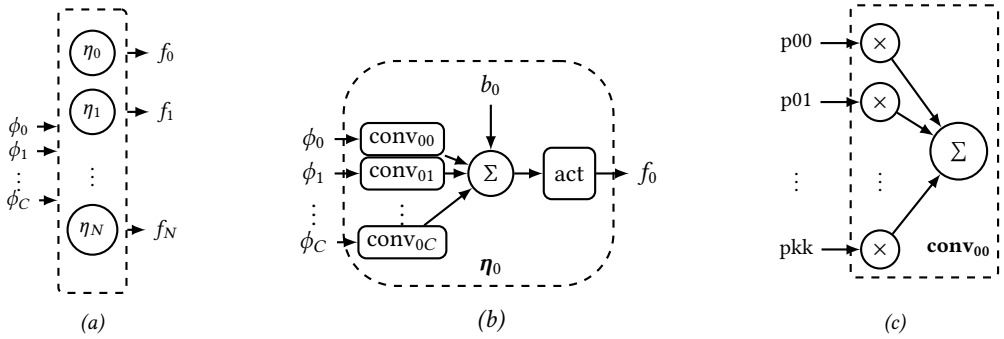

Figure 2: The 3 levels of DHM implementation of CNN entities:
(a) in convolution layers, (b) in neurons, (c) in convolution engines

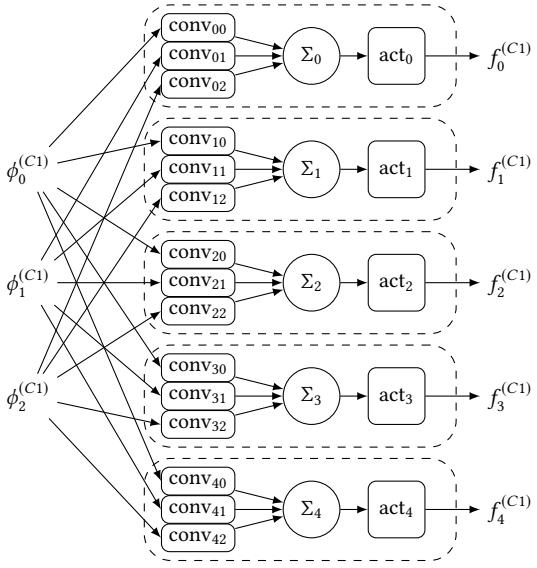

Figure 3: Applying the 3 levels of DHM (fig 2) to a dummy convolutional layer C1 (N=5, C=3, K=3):
15 separate convolution engines (135 Multipliers and 15 adders) plus 5 adders and 5 activation blocks are required to process the layer in a full parallel fashion. (bias omitted)

4 Optimizing DHM-based CNN accelerators

Direct Hardware Mapping of CNNs completely removes the need for an external memory to store intermediate results or parameters. Moreover, thanks to the fully pipelined execution model, the global throughput is only limited by the maximum clock frequency. However, these advantages come at the cost of a high resource consumption since the whole graph has to mapped onto the physical resources of the FPGA. In certain cases, this could limit the complexity of the CNNs that can be handled by the DHM approach. It is crucial, therefore, to ensure that the core operations involved in CNN actors can be translated efficiently in hardware. The most important issues, by far, are those related to on-chip memory requirements on one hand, and the implementation of arithmetic operators on the other hand.

4.1 Neighborhood extraction

Dataflow-based processing of convolutions –such in [24]– can be divided into 2 parts: neighborhood extraction (NE) and Multiply-ACCumulation (MAC).

Neighborhood Extraction (NE) relies on buffers to grant a full access to the $K^{(l)} \times K^{(l)}$ neighbors of each pixel (as shown in figure 4). Such an architecture is advantageous since it can directly extract the neighborhood of streams of pixels each clock-cycle.

Multiply Accumulate (MAC) performs a multiplication of neighborhood pixels with pre-learned kernels then accumulates the result to output feature maps. As long as the access to full neighborhood pixels is guaranteed, each of the multiplications of can be performed in a parallel way using $K^{(l)} \times K^{(l)}$ multipliers (as shown in Fig 2-c). Combining NE and parallel MAC strategy fully exploits the intra Kernel parallelism of CNNs which grants high acceleration to convolutions and, consequently, the feature extraction process.

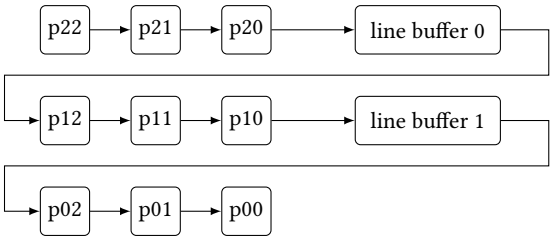

Figure 4: Architecture of a 3×3 neighborhood extractor : 2 Buffers with image length size are required to perform a 3×3 convolution on streams of pixels p_{ij}

4.2 Neighborhood Extraction Factorization (NEF)

When adopting a DHM approach, it is possible to factorize the neighborhood extraction process in order to optimize the memory print of convolutional layers. In this case, it is possible to rely only on on-chip memory buffers to process a hole convolutional layer.

Thus, since multiple neurons in a given layer have same input features to process (only the convolution kernels change), the neighborhood extraction entity can be factorized for each input feature map. This will divide the memory requirements of each layer by a factor $N^{(l)}$ (cf figure 5). For instance, the first layer of the AlexNet CNN (N=96,C=3,K=11) would require $96 \times 3 \times 11 \times 11 = 34KB$ of buffer memory to be processed, while a factorization of neighborhood extractors leads to 96 times less memory requirements ($0.3KB$). Full results of NEF on Alexnet layers are detailed in figure 6.

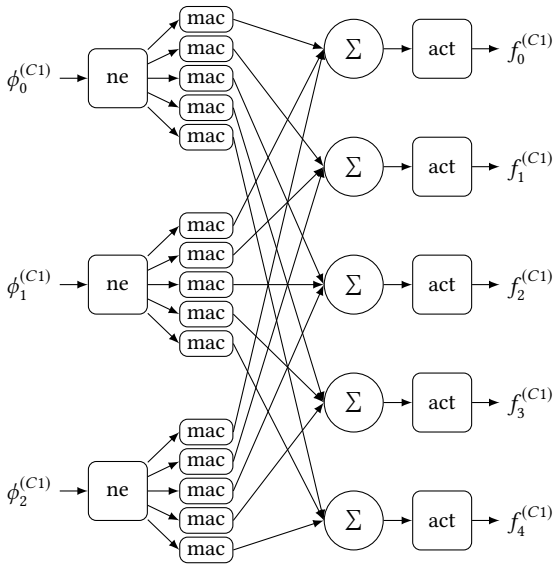

Figure 5: Data-path of a convolutional layer (bias omitted): The factorization of neighborhood extraction process reduces the memory buffers by a factor of 5 when compared to figure 3

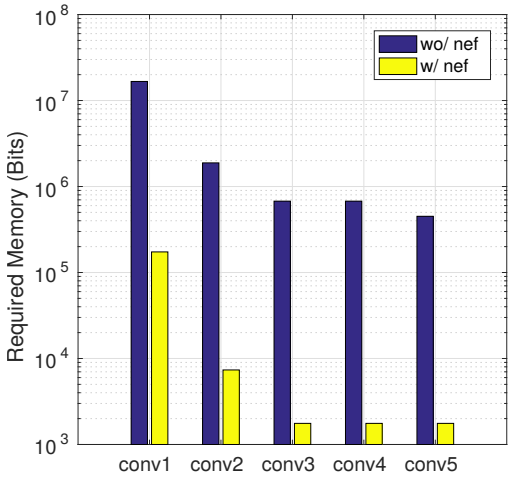

Figure 6: Ratio of memory requirements between architectures w/ and wo/ NEF for Alexnet convolutional layers: 390% less memory is required when factorizing the neighborhood extractors

4.3 Constant multiplication

4.3.1 Fixed-point computing for CNNs

Several studies [13, 14] have demonstrated that CNNs, and more generally deep learning applications, usually tolerate approximate computations with short fixed-point arithmetic. Frameworks such as Ristretto [14], for example, can perform fine-tuning of data representation in order to support fixed-point numerical representations with variable data lengths. In particular, an 8-bit (resp. 2-bit) precision is sufficient to infer the AlexNet [17] (resp. LeNet [19]) CNNs with little to no degradation in classification accuracy. The DHM approach advocated in this paper can indeed take advantage of this to significantly reduce the amount of required hardware resources by first inferring the minimal required precision and then *deriving* the size of the hardware resources to exactly match this precision.

4.3.2 Multiplications with Logic Elements

Convolutions require many multiplications. If these multiplications are implemented using hardwired Digital Signal Processing (DSP) blocks within the target FPGA, this dramatically limits the complexity of the CNN that can be implemented. For instance, the second layer of the LeNet5 network ($C = 6, N = 16, K = 5$) requires 2400 multipliers. This number largely exceeds the number of DSP blocks provided by many FPGAs and, especially by embedded devices. We overcome this problem by systematically forcing the synthesis tool to implement multiplications with logical elements instead of DSP blocks, leading the resulting implementations to rely on AND gates and trees of half-adders [2].

In addition, we take advantage of the fact that in the case of CNNs the convolution kernels – and hence the second operand of most of multiplications – are actually constants (derived from the offline training stage). It is therefore possible to use a specialized version for those multiplier instances. While this approach limits the flexibility of the system – it requires to re-compile and re-synthesise the VHDL design whenever parameters values are changed –, it delegates to the synthesis tool the task to perform low-level area and performance optimizations. More particularly, multiplications by 0 (*resp* 1) are removed (*resp.* replaced by a simple signal connection) and multiplications by a power of 2 are implemented using shift registers.

	Multiplicand	
	Variable	Constant
LE Based	ALM: 380 (0.67 %)	ALM : 121 (0.21 %)
	DSP : 0 (0 %)	DSP : 0 (0 %)
DSP Based	ALM : 71 (0.12 %)	ALM : 70 (0.12 %)
	DSP : 10 (6.41 %)	DSP : 7 (4.48 %)

Table 1: Resource utilization of a random 3×3 convolution engine on an Altera Cyclone V device with different implementations.

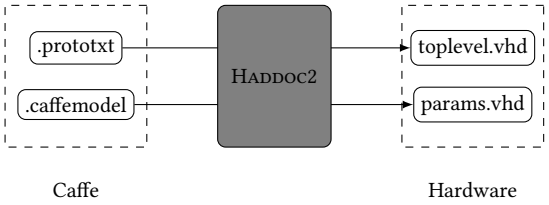

Figure 7: Hardware generation: the CNN layer arrangement is described in the top-level files while kernel parameter values and layer specification are written on the configuration file.

5 The HADDOC2 utility

The HADDOC2 framework is set of tools built upon the principles and optimization techniques described in the previous section. It is capable of automatically generating a platform independent hardware description of a CNN from a Caffe model [16]. First, layer specifications (Layer type, Number of input channels C, Number of output features N, kernel size K) are extracted from the Caffe model and the learned parameters are read, rounded to a fixed-point representation format and written as generic parameters in a configuration file. Second, a top-level VHDL file is created by transforming the dataflow graph described in Caffe. The top-level instantiates a set of generic layers parametrized according to the Caffe model specifications. These layers are described using a small number of basic predefined actors. These actors, written in a structural VHDL, follow the dataflow execution semantics discussed in the latter sections. The output is a platform independent VHDL code that can be implemented on the FPGA device using the adequate synthesis tool. The HADDOC2 framework and the library of CNN actors supporting the DHM approach are open-source and available. online[1].

Listing 1: Caffe description of a *conv* layer

```
name: "LeNet"

...

layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

...
```

Listing 2: Generated VHDL code of the layer

```
...
architecture RTL of lenet is

...

conv2: convLayer
  generic map(
    PIXEL_SIZE   => PIXEL_SIZE,
    IMAGE_WIDTH  => CONV2_IMAGE_WIDTH,
    KERNEL_SIZE  => CONV2_KERNEL_SIZE,
    NB_IN_FLOWS  => CONV2_IN_SIZE,
    NB_OUT_FLOWS => CONV2_OUT_SIZE,
    KERNEL_VALUE => CONV2_KERNEL_VALUE,
    KERNEL_NORM  => CONV2_KERNEL_NORM,
    BIAS_VALUE   => CONV2_BIAS_VALUE
  )
  port map(
    clk      => clk,
    reset_n  => reset_n,
    enable   => enable,
    in_data  => pool1_data,
    in_dv    => pool1_dv,
    in_fv    => pool1_fv,
    out_data => conv2_data,
    out_dv   => conv2_dv,
    out_fv   => conv2_fv
  );

...
```

[1] https://github.com/KamelAbdelouahab/haddoc2

6 Experimental Results with Haddoc2

As a proof of concept, we have implemented, using the Haddoc2 framework, FPGA-based accelerators for three CNN-based applications, listed in Table 2. The first one is the Caffe version of the LeNet5 [19] CNN that requires 20.78 MOPs to process a frame of size 28x28. The second application is the face detector used in [12] which requires 622.08 MOPs to process a 320x240 frame. The last one is introduced in [15] to perform car type classification and requires 268.28 MOPs to process 96x96 frames. The two first CNNs have been trained using Caffe while the third model has been directly downloaded as a Caffe pre-trained model. Table 2 gives parameter values for each CNN convolutional layer. LeNet5 and CarType CNNs have 2 convolutional layers while FaceDetect has 3. The corresponding hardware descriptions of each network have been automatically generated using Haddoc2 on an Intel i7-4770 CPU and were synthesised on two FPGA devices using respectively Intel Quartus 16.1 and Xilinx Vivaldo 2016.4.

Table 2: Topology of the convolutional layers of studied CNNs.

	LeNet5 [19]			FaceDetect [12]			CarType [15]		
Input size	28 x 28			320 x 240			96 x 96 x3		
Layer parameters	N	C	K	N	C	K	N	C	K
conv1+maxpool	20	1	5	6	1	7	32	3	5
conv2+maxpool	50	20	5	10	6	7	32	32	5
conv3	–	–	–	30	10	3	–	–	–
Kops/Pixel	26.5			6.3			29.1		

Table 3 reports post-fitting results of the LeNet-5 accelerator on an embedded Intel Cyclone V 5CGXFC9E7 device using 3 implementation strategies. In the first case, only DSP blocks are used to map the CNN multiplications. The resulting hardware requires 72× the available resource of the device. The second case features an implementation of multiplication based on logic elements and requires 3.8× the available logic. Using tailored multipliers reduces resources by a factor of 8.6×, fitting the CNN accelerator onto an Intel Cyclone V device.

Table 3: Resource utilization by a DHM LeNet5 CNN with different implementations strategies for multipliers.

	DSP-based	LE-based	LE-based + const.
Logic Usage (ALM)	NA	433500 (381%)	50452 (44%)
DSP Block usage	24480 (7159 %)	0 (0%)	0 (0%)

Table 4 details post fitting results on two embedded FPGA platforms: the Intel Cyclone V 5CGXFC9E7 and the Xilinx Kintex7 XC7Z045FBG. To the best of our knowledge, these numbers are the first to demonstrate the applicability of a DHM-based approach for the implementation of CNNs on embedded FPGAs. The three hardware accelerators fit onto the embedded devices with no off-chip memory requirement. The memory footprint shown in post fitting reports corresponds to line buffers used by the dataflow-based convolution engine and both synthesis tools instantiate LUT-based memory blocks to implement these buffers. As expected when using DHM, the logic utilization in the FPGA grows with the the topology of the CNN. However, in all the studied cases, the resources are sufficient to support direct hardware mapping. Finally, the same table reports timing analysis results of the three generated hardware accelerators. With a peak frequency of 62.3 MHz for the CarType CNN, DHM grants a maximum computation throughput of 1813

GOPs/s. For the face detection neural network, the presence of a third convolutional layer in the pipeline drops the maximum frequency to 56.7 MHz (i.e 357 GOPs/s) in the Cyclone device, which corresponds to 164 classifications/sec on 512x512 images with a 3-multiscale pyramid.

Table 4: Resource Utilization of the Haddoc2-generated convolutional layers of studied CNNs with 5-bit representation on: a- an Intel Cyclone V FPGA, b- a Xilinx Kintex 7 FPGA.

		LeNet5 [19]	FaceDetect [12]	CarType [15]
a	Logic Elements (ALMs)	50452 (44%)	6158 (5%)	48243 (42%)
	DSP Blocks[1]	0 (0 %)	0 (0%)	0 (0%)
	Block Memory Bits	2752 (1%)	41408 (1%)	28320 (1%)
	Frequency	69.14 MHz	56.7 MHz	66.0 MHz
	Processing capabilities	1832 GOPs/s	357 GOPs/s	1920 GOPs/s
b	Slices	48114 (88%)	6221 (11%)	49082 (89%)
	DSP Blocks[1]	0 (0%)	0 (0%)	0 (0%)
	LUTs as Memory	420 (1%)	1458 (2%)	1154 (1%)
	Frequency	62.13 MHz	44.41 MHz	62.3 MHz
	Processing capabilities	1646 GOPs/s	279 GOPs/s	1813 GOPs/s

7 Related work

Several studies leverage on FPGA computational power and hardware flexibility to implement the feed-forward propagation of CNNs. A non exhaustive review of these can be found in [18]. In most of approaches, acceleration of CNN-based applications is provided by mapping a limited subset of processing elements onto the target device. This is the case for example in [21] where authors describe an accelerator for the AlexNet CNN [17] implemented on a large Stratix V FPGA which, to the best of our knowledge, outperforms most state-of-the-art implementations in terms of computational and outperformed most of state-of-the-art implementations such [4,11,22]. Most of these designs are FPGA based accelerators for convolution with a relatively similar architecture of parallel processing elements associated with embedded hardcore processors running a software layer. Other approaches like [26] relies on analytical design scheme using the roofline model and loop tiling to propose an inference engine where the attainable computation roof of the FPGA is reached. This loop tilling optimization is performed on a C code then implemented in floating point on a Virtex 7 485T using Vivaldo HLS Tool.

As it has been seen in the latter sections, feed forward propagation is an algorithm that intrinsically suits to dataflow processing. Thus, dedicated stream processors for CNNs have been proposed. The most notable contribution was neuFlow [10]: A runtime reconfigurable processor for real-time image classification. In this work, Farabet and al. introduced a grid of processing tiles that were configured on runtime to build a dataflow graph for CNN applications. It was associated to "luaFlow": a dataflow compiler that transforms a high-level flow-graph representation of an algorithm (in a Torch environment [6]) into machine code for neuFlow. Such architecture was implemented on a Virtex 6 VLX240T and provided a 12 fps categorization for 512x375 images. Thus, NeuFlow transformed a CNN graph into a set of dataflow instructions, where each instruction is described as an hardware configuration of 2D-processing elements called *Processing tiles (PTs)*. Execution of the graph is carried out by sequencing the instructions on the target FPGA. This approach requires an external memory to store intermediate results, which in turn, even

with the help of a DMA, limits the final speedup.

By contrast, the DHM approach and HADDOC2 tool introduced in the present work performs all processing *on the fly* and does not require an external memory to store intermediate results. Throughput is therefore not limited by off-chip memory bandwidth. Previous works in [1] describe a first version of Haddoc that relied on the Caph [23] , a High-Level Synthesis (HLS) tool to provide dataflow-based hardware accelerators for CNNs on FPGAs. While this implementation operated at very high frame-rates (800 classifications/sec on 256×256 images), the over-head that comes with the HLS heavily restrained the size of CNNs to be implemented.

Bibliography

[1] Abdelouahab, Bourrasset, Pelcat, Berry, Serot, and Quinton. A Holistic Approach for Optimizing DSP Block Utilization of a CNN Implementation on FPGA. In *ICDSC*. ACM, 2016.

[2] Altera. Implementing Multipliers in FPGA Devices, Application Note. Technical report, 2004.

[3] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *Arxiv*, 2016.

[4] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. *ACM- SIGARCH Comput. Archit. News*.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0, 2014.

[6] R Collobert. Torch. NIPS Workshop on Machine Learning Open Source Software, 2008.

[7] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International Conference on Artificial Neural Networks*, pages 281–290. Springer, 2014.

[8] Jia Deng, Wei Dong, Richard Socher, and Liand al. Imagenet: A large-scale hierarchical image database. In *CVPR 2009. IEEE Conference*.

[9] Jack B Dennis and David P Misunas. A Preliminary Architecture for a Basic Data-flow Processor. ISCA '75. ACM.

[10] C Farabet, Yann LeCun, Eugenio Culurciello, and B Martini. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPRW'11,IEEE Computer Society Conference*.

[11] C Farabet, C Poulet, J Y Han, and Y LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *FPL International Conference on*, 2009.

[12] Clement Farabet, Cyril Poulet, and Yann Lecun. An FPGA-Based Stream Processor for Embedded Real-Time Vision with CNNs. 2009.

[13] Suyog Gupta, Ankur Agrawal, and al. Deep Learning with Limited Numerical Precision. JMLR Conference Proceedings, 2015.

[14] Philipp Gysel, Mohammad Motamedi, and all. Hardware-oriented Approximation of Convolutional Neural Networks. *Iclr*, 2016.

[15] Heikki Huttunen, Fatemeh Shokrollahi Yancheshmeh, and Chen Ke. Car type recognition with Deep Neural Networks. *IEEE Intelligent Vehicles Symposium, Proceedings*, 2016-August:1115–1120, feb 2016.

[16] Yangqing Jia, Evan Shelhamer, and al. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia*, 2014.

[17] Alex Krizhevsky, Ilya Sutskever, and Hinton Geoffrey E. ImageNet Classification with Deep CNNs. *(NIPS2012)*.

[18] Griffin Lacey, Graham WG Taylor, and al. Deep Learning on FPGAs: Past, Present, and Future. *Arxiv*, 2016.

[19] Y LeCun, L Bottou, Y Bengio, and all. Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[20] Mohammad Motamedi, Philipp Gysel, and Aal. Design space exploration of FPGA-based Deep CNNs. In *2016 (ASP-DAC)*.

[21] Kalin Ovtcharov, Olatunji Ruwase, and al. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. 2015.

[22] M Peemen, A Setio, B Mesman, and H Corporaal. Memory-centric accelerator design for Convolutional Neural Networks. In *ICCD, 2013 IEEE*.

[23] J Sérot and F Berry. High-Level Dataflow Programming for Reconfigurable Computing. In *Computer Architecture and High Performance Computing Workshop*, 2014.

[24] Richard G Shoup. Parameterized convolution filtering in a field programmable gate array. In *Oxford, United Kingdom: Abingdon EE&CS Books*. Citeseer, 1994.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, pages 1–14, 2014.

[26] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, FPGA, pages 161–170, 2015.